Inductive Logic Programming Beyond Logical Implication

Jianguo Lu*
Institute for Social Information Science, Fujitsu Laboratories Ltd.
Department of Computer Science, Fudan University

Jun Arima†
Institute for Social Information Science, Fujitsu Laboratories Ltd.

Abstract

This paper discusses the generalization of definite Horn programs beyond the ordering of logical implication. Since the seminal paper on generalization of clauses based on θ subsumption, there are various extensions in this area. Especially in inductive logic programming(ILP), people are using various methods that approximate logical implication, such as inverse resolution(IR), relative least general generalization(RLGG), and inverse implication(II), to generalize clauses. However, the logical implication is not the most desirable form of generalization. A program is more general than another program does not necessarily mean that the former should logically imply the latter. Instead, a more natural notion of generalization is the set inclusion ordering on the success set of logic programs. We observe that this kind of generalization relation is especially useful for inductive synthesis of logic programs. In this paper, we first define an ordering between logic programs which is strictly weaker than the implication ordering. Based on this ordering, we present a set of generalization rules borrowed from unfold/fold program transformation method and ILP. We also give some strategies to apply those rules.

1 Introduction

This paper discusses the problem of synthesis of logic programs from a small set of positive random examples. Since the seminal paper on generalization of clauses based on θ subsumption[Plotkin70] [1], there are various extensions

*Address: Department of Computer Science, Fudan University, Shanghai 200433, P. R. China. Email: jglu@ms.fudan.sh.cn.

†Address: Institute for Social Information Science, Fujitsu Laboratories Ltd., 140, Miyamoto, Numazu-shi, Shizuoka 410-03, Japan. Email: arima@iias.flab.fujitsu.co.jp.

[1] It is defined as: given two clauses C_1 and C_2, C_1 θ subsumes C_2 if there exists a substitution σ such that $C_1\sigma \subseteq C_2$.

in this area. Especially in inductive logic programming(ILP), it is extended at least in the following two dimensions. One is to extend from comparing two single clauses to two clauses with background theory[Plotkin71][MF], and to two programs[Buntine].

Another dimension is to extend the ordering to be considered, i.e., in what sense an object is more general than another object. Two of the extremes are θ subsumption and logical implication. The weakness of the θ subsumption is that it goes up too quickly along the generalization hierarchy. The strength is that it has good properties such as the θ subsumption ordering in clauses forms a lattice. That means for any two clauses, the least general generalization exists and is unique. In addition, we have efficient methods to compute the least general generalization. The weakness and the strength of the logical implication ordering is just the opposite. Hence, lying between the two extremes there are a spectrum of orderings being investigated. The inverse resolution(IR)[MB], and saturation[RP] are two of the examples. Especially, some people argued that the notion of implication is the most desirable form of generalization since the concept of an inductive conclusion is defined in terms of logic consequence. Consequently, lots of efforts are devoted to the investigation of orderings that approximate the implication ordering [ALLM, MF, ALLM, MF, Idestam-Almquist93].

However, generalization under implication ordering is not satisfactory in two aspects. Conceptually, the word generalization does not entail that only the implication ordering (or some ordering stronger than implication) should be used. Another choice is the set inclusion ordering under semantics of program. In practice, a program is more general than another program does not necessarily mean that the former should logically imply the latter. There are many cases that generalization under implication relation is not adequate. To illustrate this, we have the following example:

Example 1 *Suppose we have the Examples E1, E2, and E3 as follows.*

```
B:  parent(a,b).
    parent(c,d).
    parent(e,f).
    grandparent(X,Y):-parent(X,U), parent(U,Y).
    ancestor(X,Y):-parent(X,Y).
E1: ancestor(X,Y):-grandparent(X,Y).
E2: ancestor(X,Y):-parent(X,V), parent(V,Y).
E3: ancestor(X,Y):-parent(X,U),ancestor(U,Y).
```

Suppose B is in the background theory. E2(and E3) does not imply E1 under the background theory. Hence E2(and E3) can not be obtained from E1 by means of LGGθ, RLGG, inverse resolution, or inverse implication. However, under the least Herbrand semantics, E2 and E1 are equivalent, and E3 is a properly more general program than E1 in the sense that the model of E3 is a super set of the model of E1.

As this example illustrates, it is often more adequate to do generalization based on semantics of our descriptional language itself(i.e., the logic program semantics) than pure logic semantics. In other words, we need to do generalization not restricted by the implication ordering. Instead, we need to go beyond implication.

In the following sections we will first define three kinds of generalization orderings between programs instead of clauses or clauses under a background theory, and introduce the ordering $\succeq_S$ on which our generalization method is based. We will show the relationship between those orderings, and their relationship with the usual orderings. This discussion is to illustrate in what sense our generalization method goes beyond logical implication.

In section 3 we present a set of rules of generalization based on unfold/fold program transformation[TS].Those rules include both deduction and induction operations. The use of some restricted form of deduction operations is justified by the fact that they preserve the $\succeq_S$ ordering. Section 4 introduces some strategies to apply the transformation rules. Section 5 gives some examples to illustrate our approach.

2 Generalizations

Generalizations are based on some kinds of orderings. Different orderings will result in different generalizations. Before presenting our method of generalization, it is necessary to introduce various notions of orderings between programs so that we can know the ordering on which our generalization is based and its relationship with other orderings.

The programs referred in this paper are sets of definite Horn clauses. In the discussion we assume the language has potentially sufficient number of constant symbols. The immediate consequence operator T_P maps Herbrand interpretations to Herbrand interpretations. It denotes one-step deduction using program P. The function corresponding to deductions of any number of steps is denoted by $[\![P]\!]$, and is defined by $[\![P]\!](X) = \cup_{i=0}^{\infty}(T_P + Id)^i(X)$, where Id is the identity function and $(f+g)(X) = f(X) \cup g(X)$. The success set SS(P) is { A: A has a successful SLD-derivation for P }. HB denotes the Herbrand base. It is known that $[\![P]\!](\phi) = SS(P) = lfp(T_P)$. The exact definitions of the above concepts can be found in [Lloyd].

As for the notions of generalizations in logic programs, there are three layers: generalization between clauses without background theory, between clauses with background theory, and between programs [2]. They can be defined from either proof theoretic or model theoretic approach. In each layer there are some kinds of orderings, two of the most fundamental are based on θ *subsumption*, $\succeq_\theta$ [3], and *logical implication*, $\succeq_I$. They are usually defined for clauses with or without background theory. In the domain of logic programming, we have to study the ordering between programs. From the model theoretic point of view,

[2]Each is a special case of the latter layer.

[3]Although $\succeq_\theta$ is defined in terms of clauses instead of programs, it can be extended to the latter case as: For two programs P_1 and P_2, $P_1 \succeq_\theta P_2$ iff for every clause D_i in P_2 there exists a clause C_j in P_1 such that $C_j \succeq_\theta D_i$.

[M88] studied the equivalence relations between logic programs. Similarly, some orderings between programs can be defined as follows:

Definition 1 *For any two programs P_1 and P_2,*

1. *$P_1 \succeq_{T_P} P_2$ if $T_{P_1}(X) \supseteq T_{P_2}(X)$ for every $X \subseteq HB$.*
2. *$P_1 \succeq_{+} P_2$ if $\llbracket P_1 \rrbracket(X) \supseteq \llbracket P_2 \rrbracket(X)$ for every $X \subseteq HB$.*
3. *$P_1 \succeq_{*} P_2$ if $\llbracket P_1 \rrbracket(\phi) \supseteq \llbracket P_2 \rrbracket(\phi)$.*

Note that $P_1 \succeq_* P_2$ iff $SS(P_1) \supseteq SS(P_2)$. It is easy to see that $\succeq_{T_P}$, $\succeq_+$ and $\succeq_*$ are transitive and reflexive. For an arbitrary ordering $\succeq_X$, if both $P \succeq_X Q$, and $Q \succeq_X P$, then we say P and Q are equal under this ordering (denoted as $P \simeq_X Q$). We use $P \succ_X Q$ to denote $P \succeq_X Q$ and $Q \not\succeq_X P$. In the sequel, we use lgg_X to denote the least general generalization under the ordering $\succeq_X$, if it exists. An important relationship between two orderings is their relative strength. We say an ordering $\succeq_X$ is stronger than $\succeq_Y$ if whenever $P \succeq_X Q$ then $P \succeq_Y Q$, where X and Y denote arbitrary subscripts. $\succeq_X$ is strictly stronger than $\succeq_Y$ if $\succeq_X$ is stronger than $\succeq_Y$ and $\succeq_Y$ is not stronger than $\succeq_X$. A weaker ordering means more programs are involved in the generalization hierarchy, hence it will generate more specific generalization. So, generally speaking, the weaker the ordering, the more desirable of the corresponding generalization.

Theorem 1 *$\succeq_{T_P}$ is strictly stronger than $\succeq_+$, and $\succeq_+$ is strictly stronger than $\succeq_*$.*

$\succeq_{T_P}$ corresponds to a particular case of the generalized subsumption in [Buntine]. As for the correspondence between the usual notions of orderings and the above notions, we can summarize with the following theorem.

Theorem 2 *The arrows go from a stronger ordering to a weaker ordering:*

$$\begin{tikzcd}
\succeq_\theta \arrow[r] & \succeq_{T_P} \arrow[d] \\
\succeq_I \arrow[r, leftrightarrow] & \succeq_+ \arrow[d] \\
 & \succeq_*
\end{tikzcd}$$

Figure 1

A question that naturally arises is: what is the more desirable ordering based on which we can synthesize programs from a small set of random examples? As illustrated by the above figure and *Example 1*, we argue that there is no reason why the straightforward notion of set inclusion ordering between semantics of logic programs (i.e., $\succeq_*$), instead of θ subsumption or implication, should not be used as a basis for generalization.

There is one obstacle to directly using $\succeq_*$ as the basis of generalization. Often we wish to compare two programs although they use different predicate or function symbols. Also in practice, we need to discuss generalization in different languages [4]. For example, for the factorial programs P1 and P2,

```
P1: fac(1,1).
    fac(X,Y):-dec(X,U),fac(U,V),mul(V,X,Y).

P2: fac(1,1).
    fac(X,Y):-newp(X,1,Y).
    newp(1,X,X).
    newp(X,ACC,Y):-dec(X,U),mul(X,ACC,V), newp(U,V,Y).
```

We would like to consider P1 and P2 are equal, rather than that $P_2 \succ_* P_1$. Hence,

Definition 2 *($\succeq_S$) Given programs P_1, P_2. L is the subset of the underlying language which consists of only the predicates occur in both P_1 and P_2. $P_1 \succeq_S P_2$ if $SS(P_1) \cap L \supseteq SS(P_2) \cap L$.*

Theorem 3 *If $P_1 \succeq_* P_2$, then $P_1 \succeq_S P_2$. Thus, $\succeq_S$ is strictly weaker than $\succeq_I$ and $\succeq_\theta$.*

Generalization of clauses under $\succeq_\theta$ is well studied. It is known that for every two clauses, lgg_θ exists, and it is unique for reduced clauses. However, for the ordering $\succeq_I$ and $\succeq_S$, many unknowns are left [5]. Below we investigate the ordering $\succeq_S$ in some more detail. Least general generalization of any two programs under the ordering $\succeq_S$ exists, and it is not unique (even under logical implication). On the other hand, if we require both the input and the output of the generalization are single definite clauses, then the least general generalization of any two clauses under the ordering $\succeq_S$ does not exist. Another unpleasant property of $\succeq_S$ is that for every two programs (or clauses) P_1 and P_2, it is undecidable to test whether $P_1 \succeq_S P_2$. Although the ordering $\succeq_S$ seems almost intractable, it is a more natural notion of generalization. In the following of this paper we present one method to do generalization under $\succeq_S$.

3 Rules

Following [Muggleton 91], we view the generalization as a program transformation process. Given two positive examples E_1 and E_2 [6], and a background theory B, the first step of generalization is to form $B \cup \{E_1, E_2\}$ (denoted by P_0).

[4] For instance, we need to introduce (or delete) new definitions (see the next section).

[5] lgg_I does not exist even for clauses without background theory [Niblett].

[6] Here for the purpose of clarity, we only consider the case that only two positive examples are involved. The extension to more examples are straightforward.

Starting from P_0, by successively applying one of the following transformation rules, a transformation sequence $P_0, ..., P_n$ is generated.

In the set of rules presented below, both deduction (unfolding) and induction (folding and anti-unification [7]) operations are used. This is the key difference between our method and the other approaches in ILP. The use of some restricted form of deduction operation can be justified by that although it goes down the implication chain, it preserves the semantics of the logic program. This can be depicted in the following:

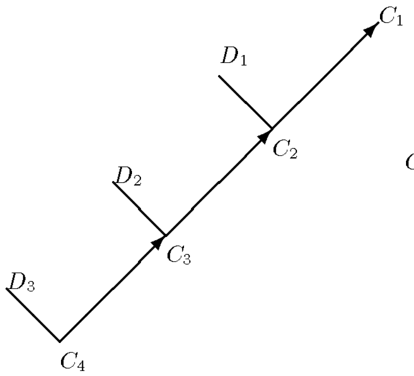

Figure 2

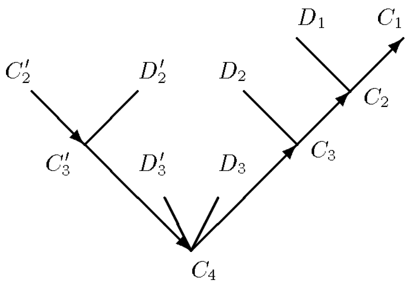

Figure 3

In the pictures above, each $C_{i+1}(C'_{i+1})$ is the result of resolution from $C_i(C'_i)$ and $D_i(D'_i)$. Current approaches perform generalizations in a manner as illustrated in figure 2: They go bottom-up along the inverse of the resolution chain (For instance, along the arrows from C_4 to C_1 in figure 2). They will not be able to tell the relationship between clauses C'_2 and C_1 in figure 3, although it may be true that $C_1 \succeq_S C'_2$. Our method allows going down the resolution chain when necessary (along the arrows from C'_2 to C_4 in figure 3), and then going up(from C_4 to C_1).

Before presenting the rules, we have some concepts to be defined.

In the following discussion we assume the programs (and examples) do not contain function symbols(i.e., programs written in Datalog). This assumption is not restrictive as we know that function symbols can always be removed by flattening [Rouveirol]. This assumption is necessary to get more specific generalizations. If some predicates are represented as function symbols instead, then some previously possible folding/unfolding operations would become impossible. This will result in more general generalizations.

Definition 3 *(nearly-matching) Suppose the lgg of C_1, C_2 under θ subsumption is C with the corresponding substitutions θ_1 and θ_2. Two clauses C_1 and C_2 are nearly-matching if $C\theta_1 = C_1, C\theta_2 = C_2$, θ_1 and θ_2 are 1-1 mappings.*

[7] In the literature the words generalization and anti-unification are often used interchangeably. Here we will use anti-unification to denote a more restricted case as defined in rule 3.

Example 2 *For the following clauses,*

```
C1: a(n1,n5) :- p(n1,n2),p(n2,n3),p(n3,n4),p(n4,n5).
C2: a(m1,m4) :- p(m1,m2),p(m3,m4).
C3: a(n1,n5) :- p(n1,n2),p(n4,n5).
```

C1 and C2 are not nearly-matching, C2 and C3 are nearly-matching.

Now we are ready to give the rules to produce generalization beyond implication. Those rules are given in the style as in [PP][TS] [8]. The unfolding and the definition rules are the same as in [PP]. Examples of the applications of these rules are in section 5.

Rule 1 *(Unfolding) Let P_k be the program $\{E_1, ..., E_r, C, E_{r+1}, ..., E_s\}$, and let C be the clause H:-F,A,G, where A is a positive literal and F and G are (possibly empty) sequences of literals. Suppose*

1. *$\{D_1, ..., D_n\}$ are all the clauses in P_j with $0 \leq j \leq k$, such that A is unifiable with $hd(D_1), ..., hd(D_n)$, with most general unifier $\theta_1, ..., \theta_n$, respectively, and*
2. *C_i is the clause $(H : -F, bd(D_i), G)\theta_i$, for $i \in \{1, 2, ..., n\}$.*

If we unfold C wrt A using $D_1, ..., D_n$ in P_j, we derive the clauses $C_1, ..., C_n$, and we get the new program $\{E_1, ..., E_r, C_1, ..., C_n, E_{r+1}, ..., E_s\}$.

The unfolding rule is essentially a deduction operation. However, it is a semantics preserving operation (i.e., $P_{k+1} \simeq_S P_k$) due to the requirement in condition 1 that the $D_1, ..., D_n$ are all the clauses that define the predicate A.

Rule 2 *(Folding) Let P_k be the program $\{E_1, ..., E_r, C_1, ..., C_n, E_{r+1}, ..., E_s\}$ and let $\{D_1, ..., D_n\}$ be a subset of clauses in program P_k, Suppose that there exists a positive literal A such that, for $i \in \{1, ..., n\}$,*

1. *$hd(D_i)$ is unifiable with A via most general unifier θ_i,*
2. *C_i is the clause $(H : -F, bd(D_i), G)\theta_i$, where F and G are sequences of literals,*
3. *$\{D_1, ..., D_n\} \cap \{C_1, ..., C_n\} = \phi$.*

If we fold $C_1, ..., C_n$ using $D_1, .., D_n$ in P_j, we derive the clause H :- F,A,G, call it C, and we get the new program $P_{k+1} \equiv \{E_1, ..., E_r, C, E_{r+1}, ..., E_s\}$.

This rule differs the usual folding rule in program transformation. Here we omit the condition that for any clause D of P_k not in $\{D_1, ..., D_n\}$, $hd(D)$ is not unifiable with A. Hence, it is a generalization operation, essentially the same as the absorption in [RP]. Here we use a more complicated form than absorption (multiple literals can be folded together) because this rule is sometimes also

[8] Here we only list a part of the relevant rules.

intended to be used as usual folding operation. In condition 2, θ_i has to be applied to the whole clause because of the multiple literal case. Condition 3 is necessary to ensure $P_{k+1} \succeq_S P_k$. A simple instance is that, without this restriction, self-folding may occur, and will result in a more specific program.

Rule 3 *(Anti-Unification) If E_1 and E_2 are nearly-matching, we may get program P_{k+1} by replacing the clauses $\{E_1, E_2\}$ in P_k by the least general generalization of E_1 and E_2 under $\succeq_\theta$.*

This restricted case of lgg_θ is used because we do not want to go up the generalization hierarchy too quickly.

Rule 4 *(Definition) We may get program P_{k+1} by adding program P_k with clauses $p(...) : -Body_i, i \in \{1, ..., n\}$, such that the predicate symbol p does not occur in $P_0, ..., P_k$.*

Here, unlike the intraconstruction[MB][RP], we have a more general rule that the body of the newly introduced clause (the $Body_i$) can be any conjunction of literals. After applying this rule, $P_{k+1} \simeq_S P_k$. Hence this is not a generalization operation. However, since the introduction of new definitions will make subsequent folding or anti-unification possible, different definitions will result in different foldings(anti-unifications), or, different generalizations. In the next section, we will give some heuristics to use this rule.

Theorem 4 *(Correctness) Let $P_0, ..., P_n$ be a transformation sequence of definite programs constructed by using the rules listed above. $P_i \succeq_S P_0$, for $i \in \{1, 2, ..., n\}$. And in general, $P_i \not\succeq_I P_0$, $P_i \not\succeq_\theta P_0$.*

The correctness of the system follows directly from the results of program transformations. Another question is the power of the system. The incompleteness is obvious, i.e., not every more general or equivalent program can be derived from given examples.

4 Strategies

Following the usual practice in program transformation, we also take the "rules + strategies" approach. Besides the usual strategies in standard program transformation[PP], here we need some additional strategies that are specific for program generalization instead of program transformation.

The general algorithm is:

Strategy 1 *(General strategy) Given a program P_j. Initially, it is P_0 which consists of background theory B and positive examples E_1 and E_2.*

1. *If anti-unification or folding is applicable for examples or newly introduced definitions, then arbitrarily do anti-unification or folding, until neither anti-unification nor folding is possible. Exit.*

2. Otherwise, use the strategies 2 or 3, go to step 1.

Strategy 2 *Unfold as much as possible until the folding or anti-unification rule is applicable, or until strategy 3 can be applied.*

The next strategy controls how to introduce new definitions. Before introducing this strategy, we need some definitions.

Definition 4 *(linking terms) Given a clause $C : H : -A_1, ..., A_m, B_1, ..., B_i$. The linking terms of the sequence of atoms $A_1, ..., A_m$ in C are the terms that occur in both $A_1, ..., A_m$ and $H : -B_1, ..., B_i$.*

Definition 5 *Given two clauses*

$C_1 : H_1 : -A_{11}, ..., A_{1m}, B_{11}, ..., B_{1i}.$
$C_2 : H_2 : -A_{21}, ..., A_{2n}, B_{21}, ..., B_{2i}.$

Suppose $C_1' : H_1 : -B_{11}, ..., B_{1i}$ and $C_2' : H_2 : -B_{21}, ..., B_{2i}$ are nearly-matching with C as their lgg_θ, and θ_1 and θ_2 as their corresponding substitutions. Suppose the number of the linking terms of $A_{11}, ..., A_{1m}$ and that of $A_{21}, ..., A_{2n}$ are equal, denote them as $t_1, ..., t_k$ and $s_1, ..., s_k$, respectively. If $\{t_l\theta_1^{-1} | \forall l \in \{1, ..., k\}\} = \{s_l\theta_1^{-1} | \forall l \in \{1, ..., k\}\} = \{X_l | \forall l \in \{1, ..., k\}\} \subseteq domain(\theta_1)$, then we say $A_{11}, ..., A_{1m}$ and $A_{21}, ..., A_{2n}$ are factors of C_1 and C_2. $\{X_l | l \in \{1, ..., k\}\}$ are called factored variables.

Note that due to the fact that here θ_1 and θ_2 are 1-1 mappings, we can have the inverse of the substitutions.

Strategy 3 *Let P_j contains the following two clauses:*

$E_1 : H_1 : -A_{11}, ..., A_{1m}, B_{11}, ..., B_{1i}.$
$E_2 : H_2 : -A_{21}, ..., A_{2n}, B_{21}, ..., B_{2i}.$

Suppose $A_{11}, ..., A_{1m}$ and $A_{21}, ..., A_{2n}$ are factors of E_1 and E_2. $X_1, ..., X_k$ are all the factored variables. Then we can have the following definition for new predicate:

$genp(X_1, ..., X_k) : -GenA_{11}, ..., GenA_{1m}.$
$genp(X_1, ..., X_k) : -GenA_{21}, ..., GenA_{2n}.$

where $GenA_{1j} = A_{1j}\theta_1^{-1}\sigma$, $GenA_{2k} = A_{2k}\theta_2^{-1}\sigma$, $j \in \{1, ..., m\}, k \in \{1, ..., n\}, \sigma$ is a 1-1 mapping with constants as its domain, new variables as its range.

This definition of genp can be added to P_j to form a new program P_{j+1}.

By folding, we can get the following program P_{j+2}:

$E_1 : H_1 : -genp(X_1, ..., X_k)\theta_1, B_{11}, ..., B_{1i}.$
$E_2 : H_2 : -genp(X_1, ..., X_k)\theta_2, B_{21}, ..., B_{2j}.$
$genp(X_1, ..., X_k) : -GenA_{11}, ..., GenA_{1m}.$
$genp(X_1, ..., X_k) : -GenA_{21}, ..., GenA_{2n}.$

This strategy shares some similarity with the intraconstruction operation in [MB][RP], and the generalization strategy in [PP]. The differences with the intraconstruction as described in [RP] are: firstly, we have a stronger applicable condition such that the newly introduced predicate has the same arity in two clauses. Secondly, we have a more general clause introduced(i.e., constants are changed into variables), so that the subsequent absorption is possible. The generalization strategy in [PP] introduces a single clause $genp() : -...$ which is a least general generalization of two clauses in an unfolding tree. The hyper least general generalization in [FIG] introduce a new predicate for two literals having the same arity.

5 Examples

It is easy to see that now we can solve the problem in example 1. Suppose the initial program P_0 is as follows. By existing methods in ILP that we are aware of, there is no way to generalize it to the desired definition of ancestor. In our method, the solution is quite simple.

Example 3 *The ancestor problem:*

```
P0: grandparent(X,Y):-parent(X,U), parent(U,Y).
    ancestor(X,Y):-parent(X,Y).
    ancestor(X,Y):-grandparent(X,Y).

P1: grandparent(X,Y):-parent(X,U), parent(U,Y).
    ancestor(X,Y):-parent(X,Y).
    ancestor(X,Y):-parent(X,U), parent(U,Y).

P2: grandparent(X,Y):-parent(X,U), parent(U,Y).
    ancestor(X,Y):-parent(X,Y).
    ancestor(X,Y):-parent(X,U), ancestor(U,Y).
```

By unfolding, we have P1 from P0. By folding, we have P2 from P1.

The next example shows the unfolding rule used in recursive definition. If we directly use the definition rule to generalize the following program, we will get a more general program.

Example 4 *The ancestor problem(continued): Suppose the initial program is P0.*

```
P0: ancestor(X,Y) :-mother(X,Y).
    ancestor(X,Y) :-mother(X,U), ancestor(U,Y).
    ancestor(X,Y) :-father(X,U), father(U,Y).

P1: ancestor(X,Y) :-mother(X,Y).
    ancestor(X,Y) :-mother(X,U), mother(U,Y).
    ancestor(X,Y) :-mother(X,U), father(U,V), father(V,Y).
    ancestor(X,Y) :-mother(X,U), mother(U,V), ancestor(V,Y).
    ancestor(X,Y) :-father(X,U), father(U,Y).

P2: ancestor(X,Y) :-mother(X,Y).
    ancestor(X,Y) :-mother(X,U), genp(U,Y).
```

```
    ancestor(X,Y) :-genp(X,Y).
    ancestor(X,Y) :-genp(X,V),ancestor(V,Y).
    genp(X,Y) :-mother(X,U), mother(U,Y).
    genp(X,Y) :-father(X,U), father(U,Y).
```

P1 is obtained from P0 by unfolding. P2 is obtained from P1 by definition introduction and folding. Now we have a program P2 which does not logically imply the program P0. If we directly use the rule *definition* to generalize the program P0, we will get a more general program P (i.e., $P \succeq_S P_2$) which logically implies P0.

```
P:  ancestor(X,Y) :-genp(X,Y).
    ancestor(X,Y) :-genp(X,V),ancestor(V,Y).
    genp(X,Y) :-mother(X,Y).
    genp(X,Y) :-father(X,U), father(U,Y).
```

Due to lack of space, here we only show how to apply the unfolding rule. The transformation rules are more powerful than it looks like. Especially, by using strategy 3, our method can effectively perform generalization under the implication ordering. For example, for programs P0 and P as below,

```
P0:ancestor(n1,n5):-parent(n1,n2),parent(n2,n3),
                    parent(n3,n4),parent(n4,n5).
   ancestor(m1,m4):-parent(m1,m2),parent(m2,m3),parent(m3,m4).
P: ancestor(X,Y):-parent(X,U),parent(U,V),parent(V,Y).
   ancestor(X,Y):-parent(X,U),ancestor(U,Y).
```

By using the structure analysis method[Idestam-Almquist95], P can be obtained from P0. By using strategy 3, we can have similar results. The difference is that here we do not need to hypothesize a new positive example according to the similarity between the structure of the two existing examples.

6 Discussions

Unlike some approaches in ILP that represent examples by a large number of ground facts, we represent the examples by a few number of clauses which embody some kind of computation trace. People may argue that the representation of positive examples such as `ancestor(c,d):-grandparent(c,d)` is quite artificial. This form of representation can be justified as follows.

- From theoretical point of view, it is desirable to study the generalization of various forms of representations.
- Our setting is automatic synthesis of logic programs from examples. From practical point of view, the number of examples required by the system should be as few as possible, and random examples should be allowed. To compensate the loss of information in limited number of examples, some kinds of computation trace (or explanation) must be given.

- It is natural to express examples in clauses instead of facts. Sometimes it is even easier than the ground fact form [9]. Especially, in the realm of programming by demonstration [Cypher], there are various methods to give traces by providing friendly user interface.

- This kind of representation may occur as an intermediate result during the process of saturating the ground facts [10]. Also, it may result from flattening a fact [11]. In this sense, our method presented in this paper is independent on our specific setting.

Our work shares some ideas in explanation based learning (EBL)[DM]. Especially, a similar ordering is used in [NMS]. In EBL, the basic steps are explanation and generalization, where the explanation is deductive derivations, and share some commonality with the unfolding operation. The difference is that here the unfolding operation has some specific requirements so that the generality is preserved. More generally, their focuses are different: we are studying the generalization between programs, while EBL focus on improving the efficiency of a problem solver.

Another work which makes a clause longer before doing generalization is saturation[RP]. This approach differs ours in that the result of saturation still logically implies the original clause.

This is an initial report on our study on the generalization beyond implication. Further work to be performed can be viewed in the following three aspects.

Study of other orderings beyond implication Although the $\succeq_S$ ordering is more satisfactory than $\succeq_I$, it is neither the unique nor the most desirable ordering beyond implication. For instance, as we might have noticed, it may be true that $P_1 \cup P_2 \succ_S lgg_S(P_1, P_2)$. This shows that sometimes lgg_S is too specific. For another instance,

Example 5 *(The non-terminating problem) Given programs P1 and P2 as follows:*

```
P1: ancestor(X,Y):-parent(X,Y).
P2: ancestor(X,Y):-parent(X,Y).
    ancestor(X,Y):-parent(X,Y),ancestor(X,Y).
```

Here $P2 \succeq_S P1$, but we won't be glad to have P2 as a generalization of P1.

So, to characterize the correct generalization in ILP, the concept of finite failure set should be incorporated in the definition of the desirable ordering.

---

[9] Consider the example: It would be easier for a user to write $fac(5) = 1*2*3*4*5$ than $fac(5) = 120$.

[10] For instance, if programs succ and multiply are defined in the background theory, then a path structure of the saturation of fac(3,2) could be
$fac(3,6) :- s(2,3), s(1,2), m(3,2,6), m(2,1,2)$. This is the same as our form of representation.

[11] For instance, [LLM] represent the positive factorial example as $factorial(sss0, sss0*(ss0*s0))$. Its flattened form is similar to our representation.

Study of the $\succeq_S$ ordering Within the scope of the study of the $\succeq_S$ ordering, we realize that the program transformation approach as presented in this paper is not the unique way. Another approach is to find a decidable ordering which approximate $\succeq_S$. We expect that for a subset of logic programs, this kind of ordering can be reduced to the θ subsumption ordering by means of function expansion.

Due to the fact that expansion (and saturation) may be infinite in general, we may need to use higher-order logic programs[Miller] to represent the result of expansion of recursive programs. Another benefit of using higher order language is that we can describe generalizations in a more formal way[Hagiya].

Study of the transformational approach Within the range of this paper, a problem is that it is hard to decide when to stop the unfolding operation. We will investigate more strategies and construct a system to do more experiments.

Acknowledgments

We would like to thank the anonymous referees for their very helpful comments. The first author wish to thank professor Masateru Harao and professor Masami Hagiya, for sharing their ideas on generalization problems, thank the Japan Society of Promotion of Science and the Education Ministry of Japan, for supporting his visits to the University of Tokyo and the Kyushu Institute of technology, and thank National Science Foundation of China, for partially supporting the research.

References

[ALLM] Aha, D. W., Lapointe, S., Ling, C. X., Matwin, S., Learning recursive relations with randomly selected small training sets. In Proceedings of the Eleventh International Machine Learning Conference (pp. 12-18). New Brunswick, NJ: Morgan Kaufmann. 1994. (NCARAI TR: AIC-94-024).

[Buntine] Wray Buntine, Generalized subsumption and its applications to induction and redundancy, Artificial Intelligence, 36(2):149-176, 1988.

[Cypher] Allen Cypher, Ed., Watch What I Do: Programming by Demonstration, The MIT Press, 1993.

[DM] DeJong, G., Mooney, R., Explanation-based generalization: an alternative view, Machine Learning, 1, 145-176, 1986.

[FIG] K. Furukawa, M. Imai, and Randy Goebel, Hyper least general generalization and its application to higher-order concept learning, draft.

[Hagiya] Masami Hagiya, Programming by example and proving by example using higher-order unification, 10th Conference on Automated Deduction (M. E. Stickel ed.), Lecture Notes in Artificial Intelligence, Vol.448, 1990, pp.588-602.

[Idestam-Almquist93] Peter Idestam-Almquist, Generalization of Horn clauses, PhD Dissertation, Department of Computer Science and Systems Science, Stockholm University and the Royal Institute of Technology, 1993.

[Idestam-Almquist95] P. Idestam-Almquist, Efficient Induction of Recursive Definitions by Structural Analysis of Saturations, in Proceedings of the Fifth Workshop on Inductive Logic Programming (ILP95), Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 1995.

[LLM] S.Lapointe, C.Ling, S.Matwin, Constructive Inductive Logic Programming, Proceedings of The Third International Workshop on Inductive Logic Programming ILP'93 April 1-3, 1993 Bled, Slovenia. 255-264.

[Lloyd] Lloyd, J.W., Foundations of logic programming, Springer-Verlag, 1984.

[MB] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In S. Muggleton, editor, Inductive Logic Programming, London, 1992. Academic Press.

[M88] Maher, M.J., Equivalence of logic programs, Foundations of Deductive Databases and Logic Programming, Morgan Kaufmann, 1988.

[MF] Muggleton, S., Feng, C., Efficient induction of logic programs. In Proceedings of the First Conference on Algorithmic Learning Theory, Tokyo., 1990. Ohmsha.

[MP] Marcinkowski, J., L. Pacholski, Undecidability of the Horn clause implication problem. Proceedings of the 33 Annual IEEE Symposium on Foundations of Computer Science, Pittsburgh, 1992. 354-362.

[MR] Muggleton, S., L. De Raedt. Inductive logic programming: theory and methods. Journal of Logic Programming, 19,20:629–679, 1994.

[Muggleton] Muggleton, S., Inverting the resolution principle. In Machine Intelligence 12. Oxford University Press, 1991.

[Miller] Miller, D., A logic programming language with lambda-abstraction, function variables, and simple unification, in Proceedings of the international workshop on Extensions of logic programming, Tubingen 1989. LCNS 475.

[Niblett] Niblett, T., A study of generalization in logic programs, In Proceedings of the third European working session on learning, Pitman, 1988.

[NMS] Numao, M., T.Maruoka, and M.Shimura, Inductively Speeding Up Logic Programs, Machine Intelligence 13, Oxford University Press 1994, pp. 371-385.

[Plotkin70] Plotkin, G. D., A note on inductive generalization, Machine Intelligence 5, Edinburgh University Press 1970, pp. 153-163.

[Plotkin71] Plotkin, G.D., A further note on inductive generalization, Machine Intelligence 6, Edinburgh University Press 1971, pp. 101-124.

[PP] Pettorossi, A., M. Proietti, Transformation of logic programs: foundations and techniques, J. Logic programming, 1994, 19(20), pp. 261-320.

[Rouveirol] Rouveirol, C., Flattening and saturation: two representation changes for generalization, Machine learning 14, pp. 219-232, 1994.

[RP] Rouveirol, C., Jean Francois Puget, Beyond inversion of resolution, in Bruce W. Porter and Ray J. Mooney(eds.) Machine learning: Proceedings of the seventh international conference on machine learning, 1990. Morgan Kaufmann. pp. 122-130.

[TS] Tamaki, H., Sato, T., Unfold/fold transformation of logic programs, in: S. A. Tarnlund (ed.), Proceedings of the 2nd international conference on logic programming, Uppsala, Sweden, 1984, pp. 127-138.